

Abstracting Complex Data Structures by Hyperedge Replacement

Stefan Rieger and Thomas Noll

RWTH Aachen University
Software Modeling and Verification Group
52056 Aachen, Germany
{rieger,noll}@cs.rwth-aachen.de

Abstract. We present a novel application of hyperedge replacement grammars, showing that they can serve as an intuitive formalism for abstractly modeling dynamic data structures. The aim of our framework is to extend finite-state verification techniques to handle pointer-manipulating programs operating on complex dynamic data structures that are potentially unbounded in their size. The idea is to represent both abstraction mappings on user-defined dynamic data structures and the (abstract) semantics of pointer-manipulating operations using graph grammars, supporting a smooth integration of the two aspects. We demonstrate how our framework can be employed for analysis and verification purposes, e.g., to prove that a procedure preserves structural invariants of the heap.

1 Introduction

Techniques for analyzing pointer programs are highly desirable. Programming with pointers is error-prone with potential pitfalls such as dereferencing null pointers and the emergence of memory leaks. When considering pointer programs we face the problem of infinite state spaces arising due to the unboundedness of the heap. Thus for employing verification methods like model checking in this scenario, abstraction techniques are indispensable.

We present an approach to abstracting state spaces of pointer programs operating on linked data structures of arbitrary size and shape. In our framework, states of the heap are modeled by hypergraphs, and both pointer-manipulating operations and abstraction mappings are represented by hypergraph transformations. More concretely we employ hyperedge replacement grammars to specify data structures and their abstractions. The essential idea is to use the replacement operations which are induced by the grammar rules in two directions. By a *backward* application of some rule, a subgraph of the heap can be condensed into a single nonterminal edge, thus obtaining an *abstraction* of the heap. By applying rules in *forward* direction, certain parts of the heap which have been abstracted before can be *concretized* again. Later we will see that this operation will be required in order to avoid the necessity for defining the effect of pointer-manipulating operations on abstracted parts of the heap.

Due to the generality of this framework, the use of hyperedge replacement grammars does not always ensure the boundedness of the resulting abstract heaps and, thus, the finiteness of the corresponding transition systems. The formalism can therefore be extended by introducing an additional parameter which allows to limit the size of the heaps. We sketch this aspect in Sect. 4.3; it is not required for understanding the actual abstraction framework.

Altogether we obtain an expressive and highly parametrized framework which allows to specify complex dynamic data structures and their abstractions in an intuitive way. Our approach is illustrated by considering a simple programming language and an example program operating on a cyclic, doubly-linked list. Using our formalism we will be able to show that the program preserves the structure of the list, independent of its size.

2 Related Work

Related work on the topic of analyzing pointer-manipulating programs can be classified into the following (often overlapping) categories: *Shape analysis* is a static analysis technique that represents recursive data structures of unbounded size by finite structures, called “shape graphs”, which are usually formalized by three-valued logical structures [6,23]. *Predicate abstraction* abstracts the state space of the program by evaluating it under a number of given predicates, obtaining a Boolean program which conservatively simulates all potential executions [3,9,19]. *Regular model checking* is a framework for unified verification of infinite-state systems based on automata theory. It represents states using words (trees) over a finite alphabet and sets of states using finite (tree) automata [7]. *Dataflow analysis* is a technique for gathering information about certain aspects of a program using its control flow graph. This approach is generally efficient but restricted to rather shallow properties of programs such as aliasing relations [17], points-to information [25], or pointer range analysis [24]. *Hoare-style approaches* extend first-order logic by reachability predicates over heap nodes [8,15]. *Separation logic* has been proposed as an extension to Hoare logic that permits local reasoning about linked structures, supporting features to support modular correctness proofs for pointer-manipulating programs [18,22].

Research in the field of *graph transformations* often concentrates on verifying and abstracting graph transformation systems, e.g. by employing so-called “Petri graphs” [4,5] or model checking state spaces generated by graph grammars [13]. We, however, *make use* of graph grammars *for* abstraction. Existing approaches with similar ideas essentially try to represent the shape of heap data structures by (abstract) graphs, and to implement statements of a programming language by graph transformation rules [20,21]. The framework presented in [1,2,10] is quite close to ours; the authors use graph reduction grammars for abstractly representing pointer structures. Their approach – which so far only handles shape safety – requires to specify an abstract transformation for each operation modifying a data structure. In contrast, we only require an abstraction specification; pointer operations do not need to be redefined in dependence of

this abstraction since they are handled automatically. Another grammar-based approach to heap abstraction is presented in [14], however, it only supports tree data structures and cannot handle DAGs and general graphs as we do.

Thus our approach is unique in that it offers a new, descriptive way for specifying abstractions on arbitrary data structures. It supports dynamic memory allocation (leading to unbounded heap sizes) and destructive updates. In addition it is easily extendable to concurrent programs with dynamic thread creation along the lines of [16].

3 Hyperedge Replacement

For the realization of our framework we concentrate on hyperedge replacement grammars [11] as they provide sufficient expressive strength for our application but still share some of the nice properties of context-free string grammars. In the following we introduce some notations that will be useful in the specification of our framework.

Given a set S , S^* denotes the set of all finite sequences (strings) over S . For $s \in S^*$ the length of s is denoted by $|s|$, the set of all elements of the sequence s is written $[s]$, and by $s(i)$ we denote the i th component of s . Given a tuple $t = (A, B, C, \dots)$ we sometimes write t_A, t_B etc. for the components if their names are clear from the context.

The domain of a function f is denoted by $\text{dom}(f)$. For two functions f and g with $\text{dom}(f) \cap \text{dom}(g) = \emptyset$ we define $f \cup g$ by $(f \cup g)(x) = f(x)$ if $x \in \text{dom}(f)$ and $(f \cup g)(x) = g(x)$ if $x \in \text{dom}(g)$. For a set $S \subseteq \text{dom}(f)$ the function $f \upharpoonright S$ is the restriction of f to S . Every $f : A \rightarrow B$ is implicitly defined on sets $f : 2^A \rightarrow 2^B$ and on sequences $f : A^* \rightarrow B^*$ by point-wise application. By $f[a/b]$ we denote the function update defined by $f[a/b](a) = b$ and $\forall c \neq a : f[a/b](c) = f(c)$. The identity function on a set S is id_S .

3.1 Hypergraphs

Hyperedge replacement grammars operate on hypergraphs, which allow hyperedges connecting an arbitrary number of vertices. Let Σ be a finite ranked alphabet where $\text{rk} : \Sigma \rightarrow \mathbb{N}$ assigns to each symbol $a \in \Sigma$ its rank $\text{rk}(a)$. We partition Σ into a set of *nonterminals* $N_\Sigma \subseteq \Sigma$ and a set of *terminals* $T_\Sigma = \Sigma \setminus N_\Sigma$. We will use capital letters for nonterminals and lower case letters for terminal symbols. We assume that both the rk function and the partitioning are implicitly given with Σ .

Definition 3.1. A (labeled) hypergraph over Σ is a tuple $H = (V, E, \text{att}, \ell, \text{ext})$ where V is a set of vertices and E a set of edges, $\text{att} : E \rightarrow V^*$ maps each edge to a sequence of attached vertices, $\ell : E \rightarrow \Sigma$ is an edge-labeling function, and $\text{ext} \in V^*$ a sequence of pairwise distinct external vertices.

We require that for all $e \in E$: $|\text{att}(e)| = \text{rk}(\ell(e))$. The set of all hypergraphs over Σ is denoted by HGraph_Σ . Furthermore we use the notations $E(v) := \{e \in E \mid v \in [\text{att}(e)]\}$ for the edges attached to a vertex and $|H| := |V| + |E|$ for the size of a hypergraph.

Thus edges are separate objects in the graph and are mapped to sequences of attached vertices. The external vertices play an important role in graph transformation steps. We will usually not distinguish between isomorphic copies of a hypergraph. Two hypergraphs H_1 and H_2 are isomorphic, written $H_1 \cong H_2$, if they are identical modulo renaming of vertices and edges.

To facilitate notation later on we introduce the notion of a *handle* which is a hypergraph consisting of only one hyperedge attached to its external nodes.

Definition 3.2. *Given $X \in \Sigma$ with $rk(X) = n$, the X -handle is the hypergraph $X^\bullet = (\{v_1, \dots, v_n\}, \{e\}, \{e \mapsto v_1 \dots v_n\}, \{e \mapsto X\}, v_1 \dots v_n) \in \text{HGraph}_\Sigma$.*

3.2 Hyperedge Replacement Grammars

Now we are ready to define hyperedge replacement grammars. They share some pleasant properties with context-free string grammars such as confluence and associativity [11], which is not the case for most other types of graph grammars.

Definition 3.3. *A hyperedge replacement grammar (HRG) over Σ is a set G of (production) rules, each of the form $X \rightarrow H$ with $X \in N_\Sigma$ and $H \in \text{HGraph}_\Sigma$ where $|ext_H| = rk(X)$.*

We denote the set of hyperedge replacement grammars over Σ by HRG_Σ and assume that there are no isomorphic production rules, i.e., rules with identical left-hand and isomorphic right-hand sides.

Fig. 1 depicts a grammar generating doubly-linked lists. The only nonterminal is the symbol D , and the letters n and p are respectively used to model the next- and previous-pointers. In the (rule-)graphs the rank of all symbols is two. The small numbers close to the connecting edges represent the order of the connected vertices and the vertices shaded in gray are the external nodes. Rules p_1 and p_2 are “redundant”; this is necessary for concretization to work (see Sect. 4.1).

The rules specify for each nonterminal X a replacement hypergraph H that will replace (the hyperedge labeled by) X when the rule $X \rightarrow H$ is applied. When a hyperedge e labeled by a nonterminal is replaced, the external vertices of the replacement graph are matched with the attached vertices of e . Thus a hyperedge replacement represents a *local* change in the graph structure.

Definition 3.4. *Let $G \in \text{HRG}_\Sigma$, $H \in \text{HGraph}_\Sigma$, $p = X \rightarrow K \in G$ and $e \in E_H$ such that $\ell(e) = X$. Let $E_{H-e} := E_H \setminus \{e\}$. We assume w.l.o.g. that $V_H \cap V_K = E_H \cap E_K = \emptyset$ (otherwise the components in K are renamed). The substitution of e by K , $J \in \text{HGraph}_\Sigma$, is defined by*

$$\begin{aligned}
 V_J &= V_H \cup (V_K \setminus [ext_K]) & E_J &= E_{H-e} \cup E_K \\
 \ell_J &= (\ell_H \upharpoonright E_{H-e}) \cup \ell_K & ext_J &= ext_H \\
 att_J &= mod \circ ((att_H \upharpoonright E_{H-e}) \cup att_K) \\
 & \text{with } mod = \mathbf{id}_{V_J}[ext_K(1)/att_H(e)(1), \dots, ext_K(rk(e))/att_H(e)(rk(e))]
 \end{aligned}$$

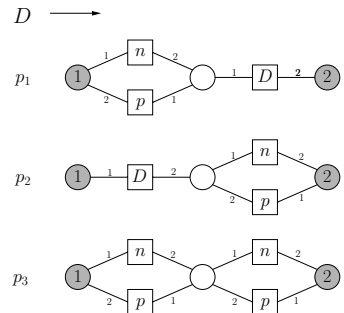


Fig. 1. HRG for Doubly-Linked Lists

We write $H \Longrightarrow_G J$ if there exist e and $X \rightarrow K$ as above. The reflexive-transitive closure and the inverse of \Longrightarrow_G are denoted by \Longrightarrow_G^* and \Longrightarrow_G^{-1} , respectively.

The *language* of a grammar $G \in \text{HRG}_\Sigma$ consists of all terminal graphs (that is, graphs that have only edges with terminal labels) that can be derived from a given starting graph $H \in \text{HGraph}_\Sigma$, i.e., $L(G, H) = \{K \in \text{HGraph}_{T_\Sigma} \mid H \Longrightarrow_G^* K\}$.

For actual applications it is important to not have nonterminals in the grammar from which no terminal graph is derivable ($\forall X \in N_\Sigma : L(G, X^\bullet) \neq \emptyset$). We call such grammars *productive*. Any HRG can be transformed into an equivalent productive grammar if its language is non-empty.

We are interested in (heap) graph abstractions for analysis and verification, which need to be effectively computable. Since, as we will see later, abstractions are obtained by backward applications of rules, the termination of the abstraction procedure can be ensured by requiring all rules in a HRGs to be *increasing*, meaning that the replacement graph (if it contains nonterminals) is “larger” than the handle of the respective nonterminal.

Definition 3.5. *A grammar $G \in \text{HRG}_\Sigma$ is increasing iff for all $X \rightarrow H \in G$ such that $\ell_H(E_H) \cap N_\Sigma \neq \emptyset$ we have $|X^\bullet| < |H|$.*

Theorem 3.6. *Let $G \in \text{HRG}_\Sigma$ be increasing and $H \in \text{HGraph}_\Sigma$. Then the set $\{K \in \text{HGraph}_\Sigma \mid K \Longrightarrow_G^+ H\}$ is finite.*

Proof. The increasingness of G implies that for any two hypergraphs H_1 and H_2 with $H_1 \Longrightarrow_G^+ H_2$ we have $|H_1| < |H_2|$ or $H_2 \in \text{HGraph}_{T_\Sigma}$. Thus for every finite hypergraph H there is a bound $n \in \mathbb{N}$ such that all derivations yielding H are of length $\leq n$ (no “loops” are possible), which proves our claim. \square

As we will see in Sct. 4 the result of Thm. 3.6 is essential for our abstraction technique since it allows us to compute a minimal abstract heap representation. Note that the HRG in Fig. 1 is increasing.

4 Abstraction of Heap States

For using HRGs as an abstraction mechanism for pointer-manipulating programs we have to represent heaps as hypergraphs. This is done by introducing two types of terminal edges: edges labeled with program variables (which we include in the terminal alphabet) are of rank one, edges of rank two – labeled with record selectors – are representing pointers in the heap. Formally, we let $T_\Sigma = \text{Var}_\Sigma \uplus \text{Sel}_\Sigma$ where $\text{rk}(\text{Var}_\Sigma) = \{1\}$ and $\text{rk}(\text{Sel}_\Sigma) = \{2\}$. Finally there are nonterminal edges of arbitrary rank that are used in the abstraction and that stand for (a set of) entire subgraphs.

Definition 4.1. *A heap configuration over an alphabet Σ is a hypergraph $H \in \text{HGraph}_\Sigma$ such that $\forall x \in \text{Var}_\Sigma : |\{e \in E_H \mid \ell_H(e) = x\}| \leq 1$ and $\text{ext}_H = \varepsilon$ where Var_Σ and Sel_Σ satisfy the constraints mentioned above. We denote the set of all heap configurations over Σ – including a special configuration H_{err}*

which is reached if pointer errors occur (i.e., dereferencing of a null pointer) – by \mathbf{HC}_Σ . A heap configuration H is called concrete if $H \in \mathbf{HC}_{T_\Sigma}$. We identify two heap configurations H and H' if $H \cong H'$.

Additional notation. For $H = (V, E, att, \ell, \varepsilon) \in \mathbf{HC}_\Sigma$, $x \in \text{Var}_\Sigma$ and $v \in V$ we write $x \hookrightarrow_H v$ to denote that $\exists e \in E : \ell(e) = x \wedge att(e) = v$. Writing $x \hookrightarrow_H \text{nil}$ is equivalent to $\nexists v \in V : x \hookrightarrow_H v$. (That is, variables pointing to *nil* are represented by omitting the corresponding edge.) For $v, w \in V$ and $s \in \text{Sel}_\Sigma$ we write $v \xrightarrow{s}_H w$ to indicate that $\exists e \in E : \ell(e) = s \wedge att(e) = vw$.

4.1 Abstraction and Concretization

When modeling the semantics of assignments it is convenient to assume that those edges which are connected to vertices that are referenced by variables, are all labeled by terminal symbols. If there is an edge e violating this property it is called a *violation point*. For all those edges we record the indices of the attached vertices that are the targets of program variables.

Definition 4.2. Let $H \in \mathbf{HC}_\Sigma$. The set of violation points $\mathcal{VP}(H) \subseteq E_H \times \mathbb{N}$ is given by:

$$\begin{aligned} & (e, i) \in \mathcal{VP}(H) \\ \Leftrightarrow & \ell(e) \in N_\Sigma \wedge (\exists x \in \text{Var}_\Sigma, v \in V_H : x \hookrightarrow_H v \wedge v = att_H(e)(i)) \end{aligned}$$

If no violation points exist a configuration is called *admissible*. As mentioned in the introduction, this will avoid the necessity for defining the effect of pointer-manipulating operations on abstracted parts of the heap.

Definition 4.3. The set of all admissible heap configurations is given by $\mathbf{aHC}_\Sigma = \{H \in \mathbf{HC}_\Sigma \mid \mathcal{VP}(H) = \emptyset\}$.

We use forward derivations to restore admissibility of a configuration. This *partial concretization* (see Def. 4.7), however, raises additional requirements for the production rules. To see this, let us again consider the example from Fig. 1. Here we could omit the rule p_2 and would still obtain a grammar that suffices to generate the language of all doubly-linked lists, thus p_2 is redundant. Omitting it, though, would lead to problems when concretizing since there might be an unbounded derivation sequence starting from a nonterminal until finally one terminal symbol is generated at its place and thus we have infinitely many concretizations. To circumvent this problem we considered to use *Greibach Normal Form* for hyperedge replacement grammars [12] (a generalization of Double Greibach Normal Form of context-free string grammars). For a HRG in Greibach Normal Form a single rule application suffices for concretization. Unfortunately this idea proved to be impractical since already for simple grammars the Greibach Normal Form is often huge [12]. Thus we decided to introduce a class of HRGs that we call *heap abstraction grammars* whose definition is admittedly more complicated.

Definition 4.4. An increasing and productive graph grammar $G \in \text{HRG}_\Sigma$ is a heap abstraction grammar if

1. $\ell_H(E_H) \cap \text{Var}_\Sigma = \emptyset$ for all $X \rightarrow H \in G$ and,
2. for every $X \in N$ with $\text{rk}(X) = k$ there exist $G_1^X, \dots, G_k^X \subseteq G$ such that
 - $\bigcup_{i=1}^k G_i^X = G$,
 - $L(G_i^X, X^\bullet) = L(G, X^\bullet)$ for all $1 \leq i \leq k$, and
 - $\ell_H(E_H(\text{ext}_H(i))) \subseteq T_\Sigma$ for all $X \rightarrow H \in G_i^X$.

The first condition disallows variables (from which we do not abstract) as edge labels. The second condition enforces a kind of symmetry for rules that have nonterminal edges connected to external vertices. The idea is to use only rules from G_i^X when concretizing a nonterminal edge from the i th attached vertex (i.e. to this vertex a variable is attached). Since we have subgrammars for all i we can concretize from any direction while avoiding “loops”. Note that the G_i^X are usually *not* disjoint.

In Fig. 1 rules p_1 and p_2 fulfill the conditions of Def. 4.4; the two rules together enable concretization from either “side” of a nonterminal edge while the generated graph language is retained. Thus, when concretizing a D -edge it suffices to apply *either* p_1 *or* p_3 (if p_1 concretizes “from the right-hand side”). For this example we have $G_1^D = \{p_1, p_3\}$ and $G_2^D = \{p_2, p_3\}$.

Based on the concepts presented so far we can formalize the notion of an abstraction function \mathfrak{A}_G , called *heap abstractor*. According to the principle that abstraction is performed by backward application of rules, \mathfrak{A}_G returns some irreducible, admissible successor of the current heap configuration with respect to the inverse derivation relation \Longrightarrow_G^{-1} .

Definition 4.5. *Let $G \in \text{HRG}_\Sigma$ be a heap abstraction grammar. A heap abstractor over G is a function $\mathfrak{A}_G : \mathbf{aHC}_\Sigma \rightarrow \mathbf{aHC}_\Sigma$ such that*

$$\mathfrak{A}_G(H) \in \{K \in \mathbf{aHC}_\Sigma \mid K \Longrightarrow_G^* H \text{ s.t. } \nexists J \in \mathbf{aHC}_\Sigma \text{ with } J \Longrightarrow_G K\}.$$

Note that heap abstraction mappings are not uniquely defined. This is only the case if \Longrightarrow_G^{-1} is confluent which, together with its well-foundedness that is implied by the increasingness of the HRG according to Thm. 3.6, yields unique normal forms. In general the abstractor should minimize the size of a heap configuration. Also note that this definition immediately implies the *correctness* of our abstraction in the sense that every concrete heap configuration can be re-generated from its abstraction:

Corollary 4.6. *Under the above assumptions, $H \in L(G, \mathfrak{A}_G(H))$ for every $H \in \mathbf{aHC}_{T_\Sigma}$.*

Since heap objects that are not reachable from program variables play no role in program semantics we delete them using a *garbage collector*. When computing the reachability of vertices we handle hyperedges of rank greater than two, i.e. nonterminal edges, conservatively as undirected edges connecting all attached vertices. We here omit the exact definition of the garbage collector and denote the mapping by $\text{GC} : \mathbf{HC}_\Sigma \rightarrow \mathbf{HC}_\Sigma$.

As already mentioned before, in addition to abstraction also concretization is necessary to restore admissibility. The essential point is that we employ *partial* concretization by applying grammar rules in forward direction. Here derivation stops as soon as the resulting heap configuration is admissible, in order to minimize the degree of concretization. Thus the properties of heap abstraction grammars as required in Def. 4.4 guarantee that only a finite number of configurations can be obtained.

Definition 4.7. Let $G \in \text{HRG}_\Sigma$ be a heap abstraction grammar and let the $G_i^X \subseteq G$ be given as in Def. 4.4. The heap concretizer, $\mathfrak{C}_G : \mathbf{HC}_\Sigma \rightarrow 2^{\mathbf{aHC}_\Sigma}$, is then defined as follows:

$$\mathfrak{C}_G(H) = \begin{cases} \mathfrak{C}_G(\{K \in \mathbf{HC}_\Sigma \mid H \Longrightarrow_{G_i^X} K\}) & \text{if } \exists (e, i) \in \mathcal{VP}(H) \wedge \ell_H(e) = X \\ \{H\} & \text{if } H \in \mathbf{aHC}_\Sigma \end{cases}$$

Note that, in contrast to a heap abstractor (Def. 4.5), the heap concretizer is uniquely defined as it yields *all* reachable (first) admissible configurations.

4.2 Pointer Programs and Their Semantics

Previously we already introduced the memory model, abstraction and concretization techniques but we still did not consider any programming language. In the following we will do this; the language itself is kept minimal to reduce the formal effort in the specification of the semantics, though it is sufficient to model most standard concepts in pointer programs.

Definition 4.8. A pointer program π is a sequence of statements $s_1; \dots; s_r$ with $s_i \in \text{CMD}$ where CMD is the set of the following commands:

$\text{PExp} ::= \text{PExp}$ (*pointer assignment*) **if** BExp **goto** n (*conditional jump*)
new(PExp) (*object creation*) **goto** n (*unconditional jump*)

Furthermore we have:

$\text{PExp} ::= \text{nil} \mid x \ (x \in \text{Var}_\Sigma) \mid x.s \ (s \in \text{Sel}_\Sigma)$
 $\text{BExp} ::= \text{PExp} = \text{PExp} \mid \text{BExp} \wedge \text{BExp} \mid \neg \text{BExp}$

Please note that for simplicity the programming language does not support arbitrary dereferencing depths. This is no restriction since this feature can be emulated by multiple assignments. An object deletion command is omitted since a *nil*-assignment with a subsequent garbage collection has the same effect. The programming language can be extended with unbounded threads and atomic regions using the concepts we introduce in [16].

In Fig. 2 an example program is shown that deletes an element from a cyclic doubly-linked list. The selectors n and p respectively model the next- and

```
delete() {
1  if  $x = \text{nil}$  goto 10;
2  if  $x = x.n$  goto 9;
3   $y := x.n$ ;
4   $x := x.p$ ;
5   $x.n := y$ ;
6   $y.p := x$ ;
7   $y := \text{nil}$ ;
8  goto 10;
9   $x := \text{nil}$ ;
10 }
```

Fig. 2. Delete from an arbitrary Cyclic Doubly-Linked List

previous-pointers. The variable x is assumed to point to some object in the structure while y is used as an auxiliary variable.

In the pointer semantics we use the special value err to denote that a pointer error (e.g. nil dereference) occurred.

Definition 4.9. For $H = (V, E, \text{att}, \ell, \text{ext}) \in \mathbf{aHC}_\Sigma$ the semantics of pointer expressions $\mathcal{P}_H[\cdot] : \text{PExp} \rightarrow V \cup \{\text{nil}, \text{err}\}$ is defined as follows:

$$\begin{aligned} \mathcal{P}_H[\text{nil}] &= \text{nil} \\ \mathcal{P}_H[x] &= v \quad \text{if } x \hookrightarrow_H v \\ \mathcal{P}_H[x] &= \text{nil} \quad \text{if } x \hookrightarrow_H \text{nil} \\ \mathcal{P}_H[x.s] &= v \quad \text{if } \mathcal{P}_H[x] \neq \text{nil} \wedge \mathcal{P}_H[x] \xrightarrow{s}_H v \\ \mathcal{P}_H[x.s] &= \text{nil} \quad \text{if } \mathcal{P}_H[x] \neq \text{nil} \wedge \nexists v \in V : \mathcal{P}_H[x] \xrightarrow{s}_H v \\ \mathcal{P}_H[x.s] &= \text{err} \quad \text{if } \mathcal{P}_H[x] = \text{nil} \end{aligned}$$

The semantics of Boolean expressions $\mathcal{B}_H[\cdot] : \text{BExp} \rightarrow \mathbb{B} \cup \{\text{err}\}$ is as usual but strict, i.e. if one of the arguments (pointer or Boolean expression) yields err the result is also err . Next we can formulate the semantics of assignments and **new**-statements, still without considering the additional steps, e.g. concretization, garbage collection and abstraction.

Definition 4.10. Let $H \in \mathbf{aHC}_\Sigma$ and $\alpha, \alpha' \in \text{PExp}$. Then we define $H[\alpha/\alpha'] \in \mathbf{HC}_\Sigma$ as follows:

- $H[x.s/\alpha'] = H_{\text{err}}$ if $x \hookrightarrow_H \text{nil}$ or $\mathcal{P}_H[\alpha'] = \text{err}$
- Otherwise we distinguish the cases given in Fig. 3 where the modifications are represented by graph transformations. Here the triangle vertex is assumed to be $\mathcal{P}_H[\alpha']$. Thus there is more than one possible result. Graph objects not shown in the source or target graphs remain unchanged.

$H[\alpha/\text{new}] \subseteq \mathbf{aHC}_\Sigma$ is given similarly by:

- $H[x.s/\text{new}] = H_{\text{err}}$ if $x \hookrightarrow_H \text{nil}$
- Otherwise the cases given in Fig. 3 apply where the triangle vertex is a new vertex inserted into V_H before applying the transformations.

Combining all the concepts introduced before we obtain the abstract heap semantics that captures the effect of the commands on the heap. For the more involved commands (assignment, **new**) the following steps are necessary:

1. execution of the actual assignment (nondeterministic)
2. garbage collection
3. partial concretization (nondeterministic)
4. re-abstraction

A fifth step may become necessary if the abstraction grammar is not suitable for the data structures occurring in the program. We then need to “artificially” bound the heap configuration by collapsing vertices. This is done by a so-called

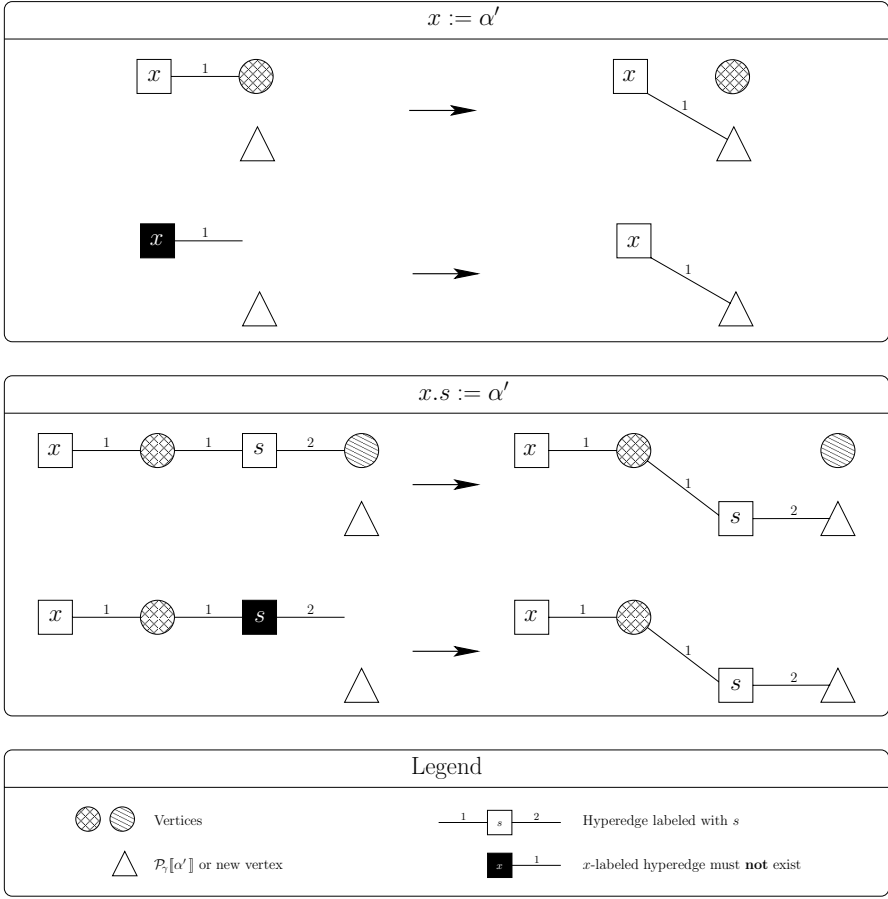


Fig. 3. Assignment / Creating Objects

heap compactor which we only briefly sketch in Sect. 4.3 in favor of concentrating on the actual abstraction.

Finally we can introduce an abstract “transition relation” that captures the effect of the statements in our programming language.

Definition 4.11. Let $G \in \text{HRG}_\Sigma$ be a heap abstraction grammar, and $\mathfrak{A}_G : \mathbf{aHC}_\Sigma \rightarrow \mathbf{aHC}_\Sigma$ a heap abstractor. The abstract heap transformation relation $\overset{h}{\Rightarrow} \subseteq (\mathbf{aHC}_\Sigma \times \text{CMD} \times \mathbf{aHC}_\Sigma)$ is given as follows for $H \in \mathbf{aHC}_\Sigma$, $H \neq H_{\text{err}}$ (we omit the **if** and **goto** statements since their semantics is straightforward and has no effect on the heap):

$$\frac{K \in \mathfrak{A}_G(\mathfrak{C}_G(\text{GC}(H[\alpha/\alpha'])))}{H, \alpha := \alpha' \overset{h}{\Rightarrow} K} \qquad \frac{K \in \mathfrak{A}_G(\mathfrak{C}_G(\text{GC}(H[\alpha/\text{new}])))}{H, \text{new}(\alpha) \overset{h}{\Rightarrow} K}$$

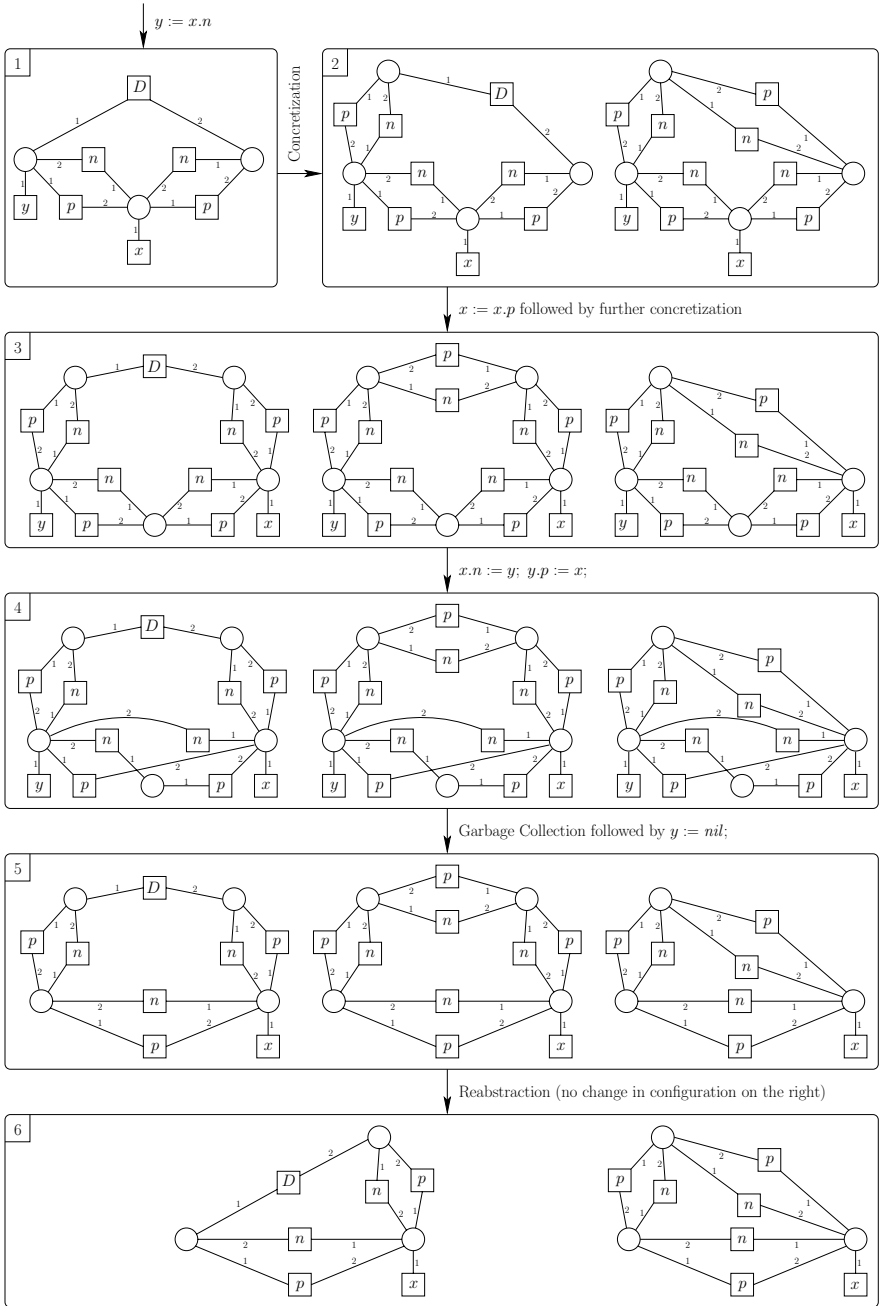


Fig. 4. Delete on Abstract Graph

Figure 4 shows the semantics of the `delete()` operation from Fig. 2 based on the HRG for doubly-linked lists (Fig. 1). We start with a configuration with three nodes and one D -edge, that is, it represents arbitrary large cycles with at least four nodes (see the left heap in subfigure 6). The `if`-statements have no effect in this case. After the first assignment $y := x.n$, we obtain the configuration depicted in subfigure 1. Here the variable y is too close to the D -edge and thus the configuration is not admissible. We have to concretize it using the rules p_1 and p_3 (from G_1^D), obtaining two resulting configurations where one only contains terminal edges (application of p_3). Rule p_2 is not applicable since it would not produce a terminal edge on the left-hand side. Note that this does not violate correctness, since the HRG from Fig. 1 is a heap abstraction grammar.

The next assignment $x := x.p$ makes a further concretization necessary since the variable x is now too close to the D -edge. Now rules p_2 and p_3 (from G_2^D) are applied to the left-hand graph from subfigure 2, and we obtain two results one of which is concrete. The third graph is resulting from the right-hand graph in subfigure 2.

The two assignments $x.n := y$ and $y.p := x$ exchange the next- and previous-pointers in the subgraph between x and y ; the result is shown in subfigure 4. The following garbage collection (the lower vertex is unreachable) and the `nil`-assignment to y yield the states visualized in subfigure 5. A re-abstraction applying rules p_1 to the left-hand and p_3 to the middle graph leads to the same result. The left-hand heap in subfigure 6 is again the initial graph, and the right-hand is the concrete one that results from deleting one node in the (minimal) cyclic list with four elements. (It is the same as in subfigure 5 since no rule is applicable).

Hence we just proved that `delete()` preserves the structure of cyclic doubly-linked lists of arbitrary size. The heap compactor is not required for our example since all abstract configurations have less than five nodes. For an insert-operation one could easily give a similar proof and would obtain even less configurations (due to the lower degree of nondeterminism).

The correctness proof for our abstraction technique requires to first define the transformation relation on concrete heaps, which is straightforward, and then to relate concrete and abstract computations in the following way. Whenever a concrete heap $H \in \mathbf{aHC}_{T_\Sigma}$ is transformed into $H' \in \mathbf{aHC}_{T_\Sigma}$ and its abstraction $\mathfrak{A}_G(H) \in \mathbf{aHC}_\Sigma$ is (abstractly) transformed into $H'' \in \mathbf{aHC}_\Sigma$, then $H' \in L(G, H'')$. That is, every concrete computation has its abstract counterpart, and thus our abstraction constitutes a safe approximation of the system.

4.3 Enforcing Finiteness

This section is optional reading and gives a short overview of the *heap compactor* which may be necessary to enforce a finite state space in certain situations where unsuitable abstraction grammars are used. The cost is an inherent loss in precision. The compactor works by merging vertices to form a special *sink* vertex if the configuration exceeds a size bound given a priori. This vertex that can represent *arbitrary* subgraphs has to be considered in the semantics and yields additional nondeterminism.

Definition 4.12. A heap compactor is a function $\kappa : \mathbf{aHC}_\Sigma \times \mathbb{N} \rightarrow \mathbf{aHC}_\Sigma$. For $H, K \in \mathbf{aHC}_\Sigma$ and $k \in \mathbb{N}$, $\kappa(H, k) = H$ if $|V_H| \leq k$ and otherwise $\kappa(H, k) = K$ such that:

- $|V_K| = k$
 - $\text{sink} \in V_K \subset (V_H \cup \{\text{sink}\})$
 - $E_K = \{e \in E_H \mid [\text{att}_H(e)] \setminus \{\text{sink}\} \neq \emptyset \vee \ell(e) \in \text{Var}_\Sigma\}$
 - $\text{att}_K = \text{mod} \circ \text{att}_H$ where
- $$\text{mod} : V_H \rightarrow V_K, \text{mod}(v) = \begin{cases} v & \text{if } v \in V_K \\ \text{sink} & \text{otherwise} \end{cases}$$
- $\ell_K = \ell_H \upharpoonright E_K$

Thus the heap compactor only modifies a configuration if the abstractor (which is to be executed beforehand) does not “compress” it enough. Its purpose is to guarantee finiteness of the semantics. If the constant k is large enough, small inconsistencies as they occur often temporarily when manipulating data structures do not result in a loss of precision since the compactor does not modify configurations with at most k nodes.

In Fig. 5 the compactor is visualized by an example. The vertices shaded in gray are merged to form the *sink* vertex visualized in black. For the actual implementation of the heap compactor a heuristics that merges connected vertices (otherwise potential dependencies between independent parts of the graph are created) and those that are distant from the program variables seems promising. The latter will reduce the probability that the *sink* vertex plays a role in the program semantics.

Modifying the expression and assignment semantics is mostly straightforward and is therefore omitted here. One essentially needs to consider additional nondeterministic cases which are introduced by the sink vertex. This leads for example to a multi-valued Boolean semantics: if an expression refers to the sink vertex we cannot decide anymore whether it is true or not and thus have to consider both cases. For assignments we get a nondeterministic step if a variable references the sink vertex.

5 Conclusions and Future Work

We have presented a framework for the analysis of pointer-manipulating programs operating on arbitrary dynamic data structures. The abstraction mechanism is parametrized via a hyperedge replacement graph grammar that models the data structure(s) used in the program. We showed how the abstract states

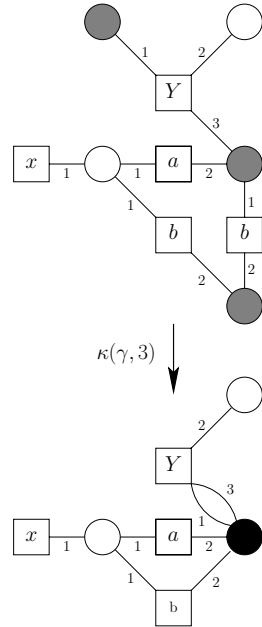


Fig. 5. Heap Compactor (Example)

can be transformed and how abstract state spaces can be generated. When employing a compactor our method ensures that these state spaces are always finite, even if the underlying data structure is outside of the specification. Smaller inconsistencies that naturally occur when manipulating data structures can be handled without loss of precision.

The programming language can be extended with concurrency, e.g. unbounded threads and atomic regions, without major changes [16]. This works essentially by modelling the control-flow semantics separately from the heap semantics by a Petri net and then combining both for state-space exploration. Hereby an orthogonal abstraction is applied on the control-flow part.

Currently we are working on an implementation of our framework. We are planning to introduce a logic to formulate verification properties and a model checking algorithm to verify those on a given program. Furthermore we will analyze how data structure definitions – as they occur in many programming languages – can be used for automatically generating an appropriate abstraction grammar.

References

1. Bakewell, A., Plump, D., Runciman, C.: Checking the shape safety of pointer manipulations. In: Berghammer, R., Möller, B., Struth, G. (eds.) *ReMiCS 2003*. LNCS, vol. 3051, pp. 48–61. Springer, Heidelberg (2004)
2. Bakewell, A., Plump, D., Runciman, C.: Specifying pointer structures by graph reduction. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) *ACTIVE 2003*. LNCS, vol. 3062, pp. 30–44. Springer, Heidelberg (2004)
3. Balaban, I., Pnueli, A., Zuck, L.D.: Shape analysis by predicate abstraction. In: Cousot, R. (ed.) *VMCAI 2005*. LNCS, vol. 3385, pp. 164–180. Springer, Heidelberg (2005)
4. Baldan, P., Corradini, A., König, B.: Verifying Finite-State Graph Grammars: An Unfolding-Based Approach. In: Gardner, P., Yoshida, N. (eds.) *CONCUR 2004*. LNCS, vol. 3170, pp. 83–98. Springer, Heidelberg (2004)
5. Baldan, P., König, B.: Approximating the behaviour of graph transformation systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) *ICGT 2002*. LNCS, vol. 2505, pp. 14–29. Springer, Heidelberg (2002)
6. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Ball, T., Jones, R.B. (eds.) *CAV 2006*. LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg (2006)
7. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: Yi, K. (ed.) *SAS 2006*. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
8. Bozga, M., Iosif, R., Lakhnech, Y.: On logics of aliasing. In: Giacobazzi, R. (ed.) *SAS 2004*. LNCS, vol. 3148, pp. 344–360. Springer, Heidelberg (2004)
9. Dams, D., Namjoshi, K.S.: Shape analysis through predicate abstraction and model checking. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) *VMCAI 2003*. LNCS, vol. 2575, pp. 310–323. Springer, Heidelberg (2002)
10. Dodds, M., Plump, D.: Extending C for checking shape safety. In: *Graph Transformation for Verification and Concurrency 2005*. ENTCS, vol. 154(2), pp. 95–112. Elsevier, Amsterdam (2006)

11. Drewes, F., Kreowski, H.-J., Habel, A.: Hyperedge replacement graph grammars. In: Rozenberg, G. (ed.) *Handbook of Graph Grammars and Computing by Graph Transformation, Foundations*, vol. I, pp. 95–162. World Scientific, Singapore (1997)
12. Engelfriet, J.: A Greibach Normal Form for Context-Free Graph Grammars. In: Kuich, W. (ed.) *ICALP 1992. LNCS*, vol. 623, pp. 138–149. Springer, Heidelberg (1992)
13. Kastenberg, H., Rensink, A.: Model checking dynamic states in GROOVE. In: Valmari, A. (ed.) *SPIN 2006. LNCS*, vol. 3925, pp. 299–305. Springer, Heidelberg (2006)
14. Lee, O., Yang, H., Yi, K.: Automatic verification of pointer programs using grammar-based shape analysis. In: Sagiv, M. (ed.) *ESOP 2005. LNCS*, vol. 3444, pp. 124–140. Springer, Heidelberg (2005)
15. Lev-Ami, T., Immerman, N., Reps, T.W., Sagiv, S., Srivastava, S., Yorsh, G.: Simulating reachability using first-order logic with applications to verification of linked data structures. In: Nieuwenhuis, R. (ed.) *CADE 2005. LNCS (LNAI)*, vol. 3632, pp. 99–115. Springer, Heidelberg (2005)
16. Noll, T., Rieger, S.: Verifying dynamic pointer-manipulating threads. In: Cuellar, J., Maibaum, T.S.E. (eds.) *FM 2008. LNCS*, vol. 5014. Springer, Heidelberg (2008)
17. Nystrom, E.M., Kim, H.-S., Hwu, W.W.: Bottom-up and top-down context-sensitive summary-based pointer analysis. In: Giacobazzi, R. (ed.) *SAS 2004. LNCS*, vol. 3148, pp. 165–180. Springer, Heidelberg (2004)
18. O’Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: *POPL 2004*, pp. 268–280. ACM Press, New York (2004)
19. Podelski, A., Wies, T.: Boolean heaps. In: Hankin, C., Siveroni, I. (eds.) *SAS 2005. LNCS*, vol. 3672, pp. 268–283. Springer, Heidelberg (2005)
20. Rensink, A.: Canonical graph shapes. In: Schmidt, D. (ed.) *ESOP 2004. LNCS*, vol. 2986, pp. 401–415. Springer, Heidelberg (2004)
21. Rensink, A., Distefano, D.: Abstract graph transformation. In: *Proc. of Int. Workshop on Software Verification and Validation (SVV 2005)*. *Electr. Notes Theor. Comput. Sci*, vol. 157(1) (2006)
22. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: *LICS 2002*, pp. 55–74. IEEE Computer Society Press, Los Alamitos (2002)
23. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24(3), 217–298 (2002)
24. Yong, S.H., Horwitz, S.: Pointer-range analysis. In: Giacobazzi, R. (ed.) *SAS 2004. LNCS*, vol. 3148, pp. 133–148. Springer, Heidelberg (2004)
25. Zhu, J., Calman, S.: Symbolic pointer analysis revisited. In: *PLDI 2004*, pp. 145–157. ACM Press, New York (2004)